

## **Gerência do Processador**

### **Bloco Descritor do processo**

Como visto na multiprogramação, processos são interrompidos e depois continuados.

Existem várias informações que o Sistema Operacional deve manter a respeito dos processos. No "programa" Sistema Operacional, um processo é representado por um registro. Esse registro é chamado bloco descritor de processo ou simplesmente descritor de processo(DP). No DP, fica tudo que o SO precisa saber sobre o processo. Abaixo, uma lista de campos normalmente encontrados no Descritor de Processo:

Prioridade do processo no sistema, usada para definir a ordem na qual os processos recebem o processador.

Localização e tamanho da memória principal ocupada pelo processo.

Identificação dos arquivos abertos no momento.

Informações para contabilidade, como tempo de processador gasto, espaço de memória ocupado, etc.

Estado do processo: apto, bloqueado, executando

Contexto de execução quando o processo perde o processador, ou seja, conteúdo dos registradores do processador quando o processo é suspenso temporariamente.

Apontadores para encadeamento dos blocos descritores do processo.

Um processo quase sempre faz parte de alguma fila. Antes de criado, o descritor do processo faz parte de uma fila de descritores livres. Após a criação, o seu descritor é colocado na fila de aptos. Normalmente essa fila é mantida na ordem em que os processos deverão receber o processador. O primeiro descritor da fila corresponde ao processo em execução. Ao fazer uma chamada de sistema associada com uma operação de I/O, o descritor do processo em execução é retirado da fila de aptos e inserido na fila associada ao periférico. O contexto de execução do processo é salvo no seu próprio descritor. Após a conclusão da operação de I/O, o seu descritor volta pra fila de aptos. Quando o processo é destruído, o descritor volta para a fila de descritores livres.

Na prática, os descritores não são copiados. Todas as filas são implementadas como listas encadeadas. A passagem do descritor de uma fila para a outra é feita através de manipulação de apontadores.

Em muitos sistemas, existe um número fixo de descritores. Ele corresponde ao número máximo de processos que podem existir no sistema. Outros sistemas utilizam alocação dinâmica de memória para criar descritores. Neste caso, não existe um limite para o número de descritores de processos. As principais características físicas do equipamento, tais como tamanho da memória principal e velocidade do processador, irão estabelecer limites quanto ao número máximo de processos. Quando alocação dinâmica de memória é utilizada na criação de blocos descritores de processos, é importante que a memória alocada fique dentro da área protegida do SO. Os descritores de processos contêm

informações vitais para a operação do sistema. Em hipótese alguma eles podem ser alterados por um processo do usuário.

Abaixo está um exemplo de descritor de processo para uma máquina hipotética simples. Suponha que o computador sendo multiprogramado possui os seguintes registradores:

- Apontador de instruções (PC)
- Apontador de pilha (SP)
- Acumulador (ACC)
- Registrador indexador (RX).

No exemplo abaixo, o sistema suporta um número máximo de processos, definido pela constante MAX\_DESC\_PROC.

```
struct desc_proc{
    char      estado_atual;          /* Estado atual do processo */
    int       prioridade;           /* Prioridade do processo */
    unsigned  inicio_memoria;       /* Endereço inicial na memória */
    unsigned  tamanho_memoria;     /* Bytes de memória ocupados */
    struct    arquivo  arq_abertos[20]; /* Arquivos abertos */

    unsigned  tempo_de_cpu;        /* Tempo já gasto de cpu */
    unsigned  proc_pc;             /* Valor salvo do reg. PC */
    unsigned  proc_sp;             /* Valor salvo do reg. SP */
    unsigned  proc_acc;            /* Valor salvo do reg. ACC */
    unsigned  proc_rx;             /* Valor salvo do reg. RX */
    struct    desc_proc *proximo;   /* Aponta para o próximo */
}

struct desc_proc tab_desc[MAX_DESC_PROC];
struct desc_proc *desc_livre;     /* Lista de descr. livres */
struct desc_proc *espera_cpu;    /* Lista de proc. esperando */
struct desc_proc *usando_cpu;    /* Aponta proc. executando */
```

Na inicialização do sistema, todos os descritores de processo podem ser encadeados, para formar uma "lista de descritores livres".

```
for(i=0; i<MAX_DESC_PROC - 1; ++i)
    tab_desc[i].proximo = &tab_desc[i+1];

tab_desc[i].proximo = NULL;
desc_livre = &tab_desc[0];
```

O apontador "espera\_cpu" indica o início da lista de processos que apontam para o processador. O descritor do processo que está executando é mantido à parte, apontado por "usando\_cpu". Outra solução é fazer com que o processo que está com o o processador ocupe a primeira posição da lista apontada por "espera\_cpu". Nesse caso, o apontador "usando\_cpu" não é mais necessário.

Para criar um processo, um descritor é retirado da lista apontada por "desc\_livre". Se "desc\_livre" contém NULL, a "lista dos descritores livres" está vazia. Nesse caso, o processo não poderá ser criado. O próximo passo é completar os campos do descritor alocado com valores apropriados. Por exemplo, o programa a ser executado pelo processo deve

ser localizado no disco, e uma área de memória grande o suficiente para ele deve ser alocada. O programa pode então ser carregado do disco para a memória principal. Essas tarefas exigem a participação dos módulos de gerência de memória e sistemas de arquivos.

Quando todos os campos estiverem preenchidos, o descritor do processo é inserido na fila "lista de espera pelo processador", apontada por "espera\_cpu". A partir desse momento, o processo passa a disputar tempo de processador. Em suma, o processo foi criado.

O procedimento de criação de processo que foi descrito é uma simplificação. Na prática, fatores como arquitetura do computador e a forma como diversos módulos do SO relacionam-se determinam o procedimento exato a ser adotado.

As operações necessárias para a destruição do processo são semelhantes. Primeiramente, todos os recursos que o processo havia alocado são liberados. Depois, o seu descritor de processo é retirado da lista em que está e inserido novamente na "lista de descritores livres". Nesse momento, o processo pára de existir. O descritor será reaproveitado na criação de um novo processo.

Para realizar uma operação do tipo "elimina todos os processos de determinado grupo", é feita uma pesquisa seqüencial sobre todos os descritores de processo em uso. Os procedimentos de destruição de processo são repetidos para todos aqueles que forem do grupo determinado.

### **Chaveamento de contexto**

Em um sistema multiprogramado, é necessário interromper processos para continuá-los mais tarde. Essa tarefa é chamada de **chaveamento de processo** ou **chaveamento de contexto de execução**. Para passar o processador do processo 1 para o 2, é necessário salvar seu contexto de execução do processo 1. Quando o P1 receber novamente o processador, o seu contexto de execução será restaurado. É necessário salvar tudo o que possa ser destruído pelo processo 2 enquanto ele executa.

O contexto de execução é formado basicamente pelos registradores do processador. O processo 2, ao executar, vai colocar seus próprios valores nos registradores. Entretanto, quando o processo 1 voltar a executar, ele espera encontrar nos registradores os mesmo valores que havia no momento da interrupção. O programa de usuário sequer sabe que será interrompido diversas vezes durante sua execução. Logo, não é possível deixar para o programa a tarefa de salvar os registradores. Isso deve ser feito pelo próprio SO.

Os conteúdos dos registradores são salvos toda vez que um processo perde o processador. Eles são recolocados quando o processo voltar a executar. Desta forma, o processo não percebe que foi interrompido. Em geral, salvar o contexto de execução do processo em execução é a primeira tarefa do SO, ao ser acionado. Da mesma forma, a última tarefa do SO é ao entregar o processador para um processo é repor o seu contexto de execução. Ao repor o valor usado pelo processo no

apontador de instruções (*program counter*), o processador volta a executar instruções do programa de usuário. O módulo do SO que realiza a reposição do contexto é chamado de **dispatcher**.

O local usado para salvar o contexto de execução de um processo é o seu próprio bloco descritor. Uma solução alternativa é salvar todos os registradores na própria pilha do processo, colocando no bloco descritor apenas um apontador para o topo da pilha. Essa solução é, em geral, mais simples e eficiente. O próprio mecanismo de atendimento de interrupções da maioria dos processadores já salva na pilha alguns registradores. Entretanto essa solução utiliza uma pilha cujo controle de espaço disponível não está mais a cargo do usuário. Se o salvamento do contexto ocorrer no momento que não existe espaço suficiente na pilha, informações pertencentes ao espaço de endereçamento do usuário poderão ser destruídas. Essa solução somente é viável quando é possível garantir que haverá espaço disponível na pilha do usuário. Uma solução de compromisso é fazer com que cada processo possua uma pilha adicional para uso exclusivo do SO. Nesse caso, primeiramente é feito um chaveamento de pilha. Depois o contexto é salvo na pilha do processo controlada pelo SO. Nessa pilha, é possível garantir que haverá espaço suficiente para o contexto de execução de processo.

## **Threads**

Um processo é uma abstração que contém uma série de atributos como espaço de endereçamento, descritores de arquivos abertos, permissões de acesso, quotas, etc. Um processo possui ainda áreas de código, dados e pilha de execução. Também é associado ao processo um fluxo de execução. Por sua vez, uma thread nada mais é do que um fluxo de execução. Na maior parte das vezes, cada processo é formado por um conjunto de recursos mais uma única thread.

A idéia de **multithreading** é associar vários fluxos de execução (várias threads) a um único processo. Em determinadas aplicações, é conveniente disparar várias threads dentro de um mesmo processo (programação concorrente). É importante notar que as threads existem no interior de um processo, compartilhando entre elas os recursos do processo, como o espaço de endereçamento (código e dados). Devido a essa característica, a gerência de threads (criação, destruição, troca de contexto, sincronização) é "mais leve" quando comparada com processos. Por outro lado, criar uma thread implica apenas definir uma pilha e um novo contexto de execução dentro de um processo já existente. O chaveamento de duas threads de um mesmo processo é muito mais rápido que o chaveamento entre dois processos. Por exemplo, como todas as threads de um mesmo processo compartilham o mesmo espaço de endereçamento, a MMU (memory management unit) não é afetada pelo chaveamento entre elas. Em função do exposto acima, threads são muitas vezes chamadas de **processos leves**.

Duas maneiras básicas podem ser utilizadas para implementar o conceito de threads em um sistema. Na primeira, o SO suporta apenas processos convencionais, ou seja, processos de uma única thread. O

conceito de thread é então implementado pelo processo a partir de uma biblioteca ligada ao programa do usuário. Devido a essa característica, threads implementadas dessa forma são denominadas **threads do nível do usuário** (user-level thread). No segundo caso, o SO suporta diretamente o conceito de thread. A gerência de fluxos de execução pelo sistema operacional não é mais orientada a processos mas sim a threads. As threads que seguem esse modelo são ditas **threads do nível do sistema** (kernel threads).

O primeiro método é denominado N:1 (many-to-one). A principal vantagem dele é o fato de as threads serem implementadas em espaço de usuário, não exigindo assim nenhuma interação com o SO. Esse tipo de thread oferece um chaveamento de contexto mais rápido e menor custo para criação e destruição. A biblioteca de threads é responsável pelo compartilhamento, entre elas, do tempo alocado ao processo. O SO preocupa-se apenas em dividir o tempo do processador entre diferentes processos. A grande desvantagem desse método é que as threads são efetivamente simuladas a partir de um único fluxo de execução pertencente a um processo convencional. Como consequência, qualquer paralelismo real disponível no computador não pode ser aproveitado pelo programa. Outro problema é que uma thread efetuando uma operação de entrada e saída bloqueante provoca o bloqueio de todas as threads do processo.

O segundo método é o 1:1 (one to one). Ele resolve os dois problemas mencionados acima: aproveitamento do paralelismo real dentro de um único programa e processamento junto com I/O. Para que isso seja possível, o SO deve ser projetado de forma a considerar a existência de threads dividindo o espaço de endereçamento no processo hospedeiro. A desvantagem é que as threads ficam "menos leves".

Existe também um terceiro método, misto, que tenta combinar as duas abordagens, chamado M:N (many-to-many). Esse método possui um escalonamento de dois níveis. Uma biblioteca no espaço do usuário seleciona as threads (M) do programa para serem executadas em uma ou mais threads do sistema (N).

## **Escalonadores**

Em qualquer SO que implemente multiprogramação, diversos processos disputam os recursos disponíveis no sistema, a cada momento. Logo, existe a necessidade de escalonar esses processos com respeito à utilização dos recursos.

No **escalonamento de curto prazo** é decidido qual processo será executado à seguir. Esse escalonador é executado com grande frequência, sempre que o processador ficar livre. Logo, deve ser mais rápido.

Em alguns sistemas, processos nem sempre são criados no momento da solicitação. Em alguns ambientes, a criação de um processo pode ser postergada se a carga da máquina estiver muito grande. Cabe ao **escalonador de longo prazo** decidir quando um processo solicitado será

efetivamente criado. Pode-se esperar a carga da máquina diminuir para então disparar um novo processo.

É sempre possível que uma sobrecarga no sistema leva ao esgotamento de recursos disponíveis. Em especial, isto pode ocorrer com a memória principal. Mesmo que isso não seja verdade no momento da criação dos processos, muitos programas alocam memória dinamicamente durante sua execução. Esse processo é chamado **swapping**. No processo swapping, considere um sistema que existam 10 processos, porém apenas 8 vagas na memória principal. A cada momento, 2 processos são completamente copiados para o disco, e os processos vão se revezando no disco. Assim, os 10 serão processados na memória de apenas 8.

Na operação **swap-out**, a execução de um processo é temporariamente suspensa e o seu código e dados são copiados para o disco. A operação **swap-in** faz o contrário.

O escalonador que decide qual processo deverá sofrer swap-in e swap-out é chamado **escalonador de médio prazo**. Esse escalonador está ligado tanto à gerência de memória quanto à gerência do processador.

O escalonador mais importante é o de curto prazo. Em geral, quando lemos o termo scheduler sem nenhum complemento, estamos lendo sobre o escalonador de curto prazo.

## **Algoritmos de Escalonamento**

Nesta seção serão vistos algoritmos para o escalonador de curto prazo. Em geral, esses mesmos algoritmos podem ser facilmente adaptados para a situação de médio ou longo prazo.

Na escolha de um algoritmo de escalonamento, utiliza-se com critério básico o objetivo de aumentar a produção do sistema, e, ao mesmo tempo, diminuir o tempo de resposta percebido pelo usuário. Esses dois objetivos podem tornar-se conflitantes em determinadas ocasiões.

Para aumentar a produção do sistema (throughput), é necessário manter o processador ocupado o tempo todo. Dessa forma, o sistema produz mais em menos tempo. Também é importante obter um baixo tempo de resposta (turnaround time) ao usuário. Isso é obtido, no caso da gerência do processador, com um baixo tempo médio de espera na fila do processador.

Para todos os algoritmos, é necessário considerar a existência de processos "I/O bound" e "cpu-bound" misturados no sistema. Também não adianta um baixo tempo médio de resposta com variância elevada. Variância elevada significa que a maioria dos processos recebe um serviço (tempo de processador) satisfatório, enquanto alguns são bastante prejudicados. Provavelmente será melhor sacrificar o tempo médio de resposta para homogeneizar a qualidade do serviço que os processos recebem.

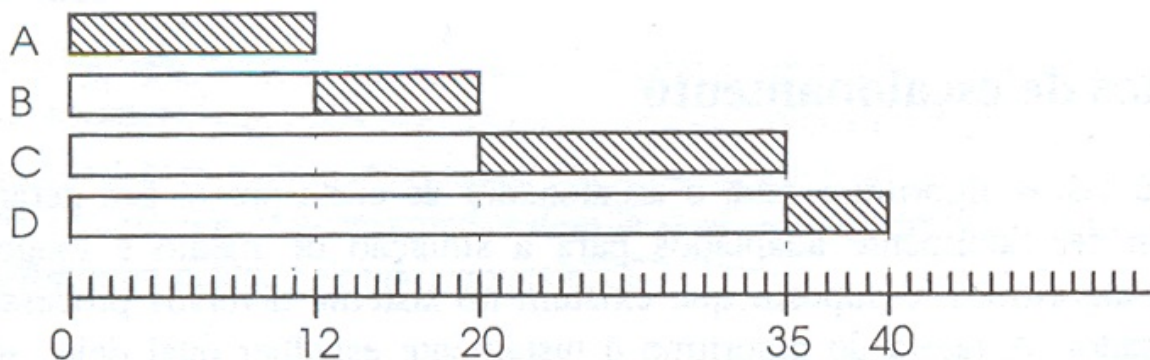
### Ordem de Chegada (FIFO – First-in first-out)

Esse é o algoritmo de implementação mais simples. A fila do processador é uma fila simples. Os processos são executados na mesma ordem que chegaram na fila. Um processo somente libera o processado quando realiza a uma chamada de sistema ou quando ocorre algum tipo de erro na execução.

O problema desse algoritmo é o desempenho. Quando um processo "cpu-bound" está na fila, todos os processos devem esperar que ele termine seu ciclo de processador e então executar.

Veja a tabela a seguir:

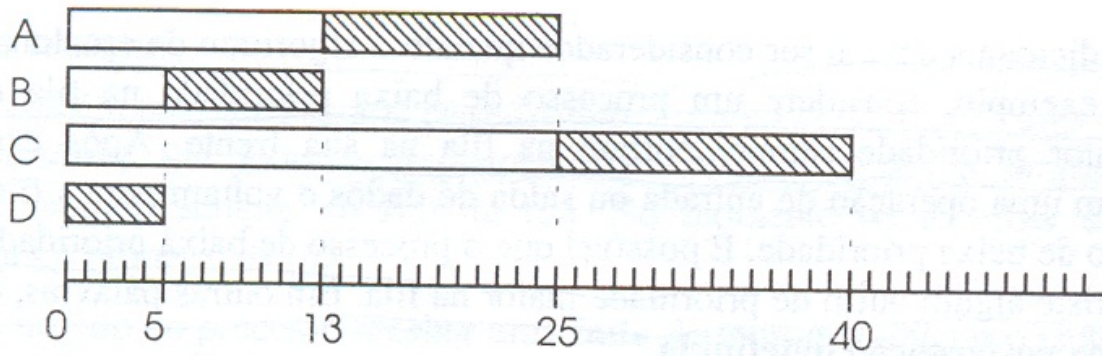
Processo	Duração do próximo ciclo de processador	
A	12	(Unidades de tempo)
B	8	"
C	15	"
D	5	"



O tempo médio de espera na fila do processador para esse conjunto de processos foi de  $(0+12+20+35)/4 = 67/4 = 16,75$ . Obviamente, se o processo "D" fosse executado antes do processo "C", obviamente o tempo médio de espera na fila seria menor. Isso justifica o próximo algoritmo, mesmo que ele não seja possível na prática.

### Ciclo de processador menor antes (SJF – Shortest job first)

O menor tempo médio de espera na fila é obtido quando é selecionado antes o processo cujo próximo ciclo de processador é o menor entre os processos que estão na fila. Esse algoritmo poderia ser implementado como uma lista ordenada na ordem crescente da duração do próximo ciclo do processador. Utilizando a mesma tabela, teremos:  $(0+5+13+25)/4 = 43/4 = 10,75$ .



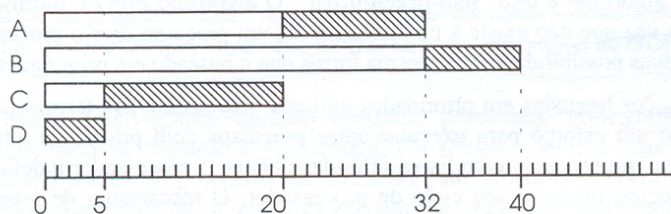
O problema desse algoritmo é que para implementá-lo é necessário prever o futuro. A duração do próximo ciclo de processador de um processo não é conhecida. Esse algoritmo, apesar de não poder ser implementado, facilita a obtenção do importante dado teórico do menor tempo de espera. É utilíssimo para criação de comparações. É importante salientar que nesse algoritmo os processos "I/O bound" são favorecidos. Isso ocorre porque eles recebem e liberam o processador rapidamente, minimizando o tempo de espera dos demais. Como regra geral, pode-se dizer que favorecer os processos de I/O diminui o tempo médio de espera na fila do processador.

## Prioridade

Quando os processos de um sistema possuem diferentes prioridades, essa prioridade pode ser utilizada para decidir qual processo é executado a seguir. Um algoritmo de escalonamento desse tipo pode ser implementado através de uma lista ordenada conforme a prioridade dos processos. O processo a ser executado é sempre o primeiro da fila. Quando dois processos possuem a mesma prioridade, algum critério para desempate deve ser utilizado. Por exemplo, a ordem de chegada (FIFO).

A prioridade de um processo pode ser definida externamente ao sistema. Por exemplo, o administrador da rede dizer que os processos dos chefes é mais importante que os processos dos funcionários. Também é possível que o próprio sistema defina a prioridade dos processos. Por exemplo, se os processos I/O bound possuírem prioridade maior, o algoritmo irá aproximar-se mais de SJF.

Processo	Prioridade	Duração do próximo ciclo de processador
A	3	12 (Unidades de tempo)
B	4	8 "
C	2	15 "
D	1	5 "



Vários aspectos adicionais devem ser considerados quando um algoritmo de escalonamento emprega prioridades. Por exemplo, considere um processo de baixa prioridade na fila do processador. Se todos os processos de alta prioridade forem executados na sua frente, é possível que este processo de baixa prioridade nunca seja executado. É o que chamamos de postergação indefinida.

Pode-se impedir a ocorrência da postergação indefinida através da adição de um algoritmo de envelhecimento (aging) ao método básico. Lentamente os processos tem sua prioridade elevada. Após algum tempo, processos que "envelheceram" sem serem executados são chamados ao processador. Lembrando sempre que o objetivo do aging não é elevar a prioridade de nenhum processo, mas sim impedir que ele fique na fila para sempre.

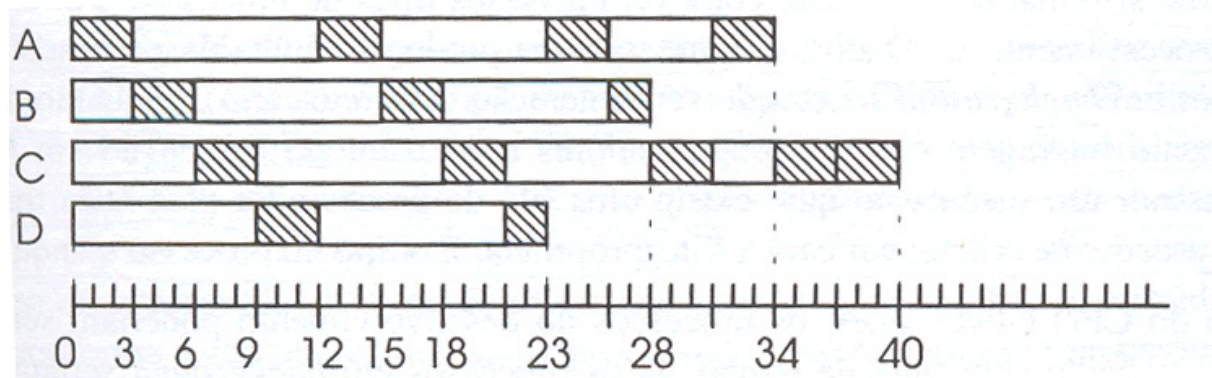
### Preempção

Um algoritmo é dito **preemptivo** se o processo em execução puder perder o processador para outro processo, por algum motivo que não seja o término do seu ciclo de processador. Se o processo em execução só libera o processador por vontade própria (chamada de sistema), então o algoritmo é dito **não-preemptivo**. O algoritmo FIFO é intrinsecamente não preemptivo, já que não existe maneira de um processo tirar outro do processador. O processo SJF admite ambas as possibilidades de preempção, assim como o processo de prioridades. Tipicamente, soluções baseadas em prioridades admitem preempção.

### Fatia de Tempo (round-robin)

Nesse método cada processo recebe uma fatia de tempo do processador (**quantum**). Ele pode ser implementado através de uma fila simples, semelhante ao FIFO. Processos entram sempre no fim da fila. No momento de escolher um processo para executar, é sempre o primeiro da fila.

A diferença está no fato do processo receber uma **fatia de tempo** ao iniciar o ciclo de processador. Se o processo realizar uma chamada de sistema e ficar bloqueado antes do término da sua fatia, , simplesmente o próximo processo da fila recebe uma fatia integral e inicia sua execução.



Um relógio de tempo real em hardware delimita as fatias de tempo através de interrupções. Nesse algoritmo não é possível postergação indefinida, pois processos sempre entram no fim da fila. Não existe como um processo ser "passado pra trás".

Um problema é definir o tamanho da fatia de tempo. É preciso levar em conta que o chaveamento entre processos não é instantâneo, como na figura acima. Ele envolve um custo em termos de tempo do processador. Se a fatia de tempo for muito pequena esse custo cresce em termos relativos. Uma fatia de tempo muito grande também apresenta problemas. Principalmente, perde-se a aparência de paralelismo na execução de processos, e para o usuário parecerá que o sistema está "dando saltos" para processar.

Quando a fatia de tempo é tão grande que todos os processos liberam o processador (fazem uma chamada de sistema) antes dela terminar, o algoritmo degrada para FIFO. Por outro lado, em sistemas onde a ilusão do paralelismo não é importante pode-se aumentar a fatia de tempo para reduzir o custo associado com chaveamento de processos.

## Múltiplas Filas

Em um mesmo sistema, normalmente convivem diversos tipos de processos. Por exemplo, num CPD, os processos de produção (folha de pagamentos, etc) podem ser disparados em background, enquanto processos do desenvolvimento atuam em foreground. É possível criar um sistema no qual existe uma fila de processador para cada tipo de processo. Quando um processo é criado, vai para a fila apropriada. É o tipo de processo que define a sua fila. Poderíamos colocar os processos em background escalonando conforme sua chegada (FIFO) e os processos de foreground com *round-robin*. Assim os processos mais atuantes teriam mais atenção e, por exemplo, à noite, os processos em background agiriam mais rapidamente.

Com múltiplas filas, o tipo do processo define a fila na qual ele é inserido, e o processo sempre volta para a mesma fila. Quando o processo pode mudar de fila durante sua execução temos múltiplas filas com realimentação. Por exemplo, pode-se utilizar esse tipo de fila para construir um algoritmo que favorece os processos I/O Bound. Para tanto, são utilizadas duas filas, A e B, cada uma trabalhando com fatias de tempo. Prioridade preemptiva é utilizada entre filas, sendo que a fila A possui prioridade maior. Quando um processo é criado, ou quando volta de uma chamada de sistema, sempre é inserido na fila <sup>a</sup> Entretanto, se o processo esgota uma fatia de tempo da fila A, ele é transferido para a fila B. No algoritmo apresentado, processos I/O Bound em geral permanecem na fila A, pois fazem uma chamada de sistema antes de esgotar a sua fatia de tempo. Já processos cpu-bound esgotam a fatia de tempo e são mandados para a fila B. Permanecendo o tempo todo na fila A, processos de I/O bound acabam recebendo um serviço melhor do que diz respeito ao tempo do processador. O sistema apresenta um "efeito-peneira" onde processos cpu-bound acabam descendo para a fila B. Os mesmos processos cpu-bound são inseridos na fila A quando voltam da chamada de sistema. Isso significa que, mesmo que um processo mude seu comportamento durante a execução, ele receberá um tratamento adequado pelo sistema.

### Finalizando

A partir dos algoritmos básicos apresentados, é possível imaginar dezenas de combinações e variações. Tipicamente, a maioria dos sistemas trabalham com fatia de tempo, também é usual trabalharem prioridade para favorecer determinados processos que realizam tarefas para o próprio Sistema Operacional. Nesses sistemas, a ilusão de paralelismo é essencial.

Para processos em background, quando não existe um usuário esperando que a resposta apareça na tela, é natural utilizar uma fatia de tempo maior. Até mesmo FIFO é viável, desde que com proteção contra laços infinitos em programas.

Por outro lado, em sistemas de tempo real que atendem a eventos externos, prioridade é essencial. Sistemas Operacionais de tempo real

devem apresentar características especiais no que diz respeito ao tempo de execução dos processos.